AD-A100 593    NAVAL POSTGRADUATE SCHOOL   MONTEREY CA                    F/G 9/2
                CONCURRENCY CONTROL OVERHEAD OR CLOSER LOOK AT BLOCKING VS. NON--ETC(U)
                JUN 81   D Z BADAL
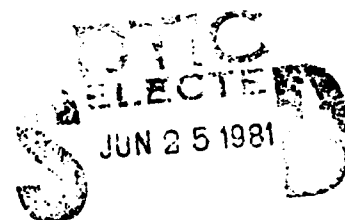UNCLASSIFIED    NPS52-81-005                                              NL

NPS52-81-005

# NAVAL POSTGRADUATE SCHOOL
## Monterey, California

CONCURRENCY CONTROL OVERHEAD OR

CLOSER LOOK AT BLOCKING

VS.

NONBLOCKING CONCURRENCY CONTROL MECHANISMS

Dusan Z. Badal

June 1981

Approved for public release; distribution unlimited

81 6 25 042

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral J. J. Ekelund                          D. A. Schrady
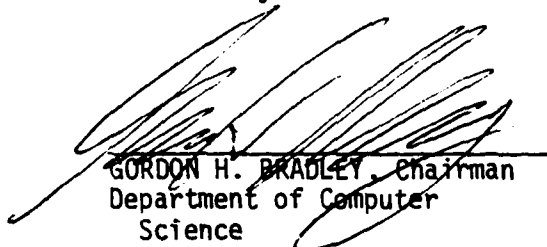Superintendent                                      Acting Provost

Reproduction of all or part of this report is authorized.

This report was prepared by:
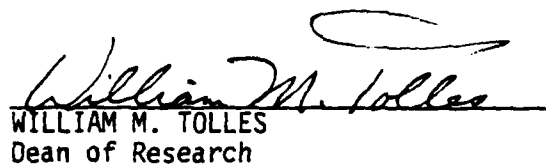
_____
DUSAN Z. BADAL
Assistant Professor of
Computer Science

Reviewed by:                        Released by:

_____   _____
GORDON H. BRADLEY, Chairman         WILLIAM M. TOLLES
Department of Computer              Dean of Research
  Science

UNCLASSIFIED

| REPORT DOCUMENTATION PAGE | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|---|
| 1. REPORT NUMBER<br>NPS52-81-005 | 2. GOVT ACCESSION NO.<br>AD-A100 593 | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (and Subtitle)<br>Concurrency Control Overhead or Closer Look at Blocking vs. Nonblocking Concurrency Control Mechanisms. | | 5. TYPE OF REPORT & PERIOD COVERED |
| | | 6. PERFORMING ORG. REPORT NUMBER |
| 7. AUTHOR(s)<br>Dusan Z. Badal | | 8. CONTRACT OR GRANT NUMBER(s) |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS<br>Naval Postgraduate School<br>Monterey, CA 93940 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS |
| 11. CONTROLLING OFFICE NAME AND ADDRESS<br>Naval Postgraduate School<br>Monterey, CA 93940 | | 12. REPORT DATE<br>June 1981 |
| | | 13. NUMBER OF PAGES |
| 14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) | | 15. SECURITY CLASS. (of this report)<br>Unclassified |
| | | 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

This report has been published in the Proceedings of the Fifth Berkeley Conference on Distributed Data Management and Computer Networks.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Distributed Databases, Synchronization, Serialization, Concurrency Control

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

In this paper we divide concurrency control (CC) mechanisms for distributed DBMS's (DDBMS) into three classes. One class consists of blocking CC mechanisms and two classes contain nonblocking CC mechanisms. We define CC overhead and derive it for conflicting and nonconflicting transactions for each class of CC mechanisms. Since CC overhead is dependent on CC mechanism only, it can be used as a metric for comparison of CC mechanisms and as a measure of CC load on DDBMS resources. We also describe two new nonblocking distributed concurrency control mechanisms which use the concept of multiple

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE
1 JAN 73
S/N 0102-014-6601 |

UNCLASSIFIED

20. data object versions. One is based on time stamp ordering of transaction execution and the other is based on nonserializable execution detection and recovery to serializable execution. We compare both with distributed two-phase locking.

Concurrency Control Overhead or Closer Look at Blocking
vs.
Nonblocking Concurrency Control Mechanisms

D. Z. Badal


Department of Computer Science
Naval Postgradute School
Monterey, CA 93940

## Abstract

In this paper we divide concurrency control (CC) mechanisms for distributed DBMS's (DDBMS) into three classes. One class consists of blocking CC mechanisms and two classes contain nonblocking CC mechanisms. We define CC overhead and derive it for conflicting and nonflicting transactions for each class of CC mechanisms. Since CC overhead is dependent on CC mechanism only, it can be used as a metric for comparison of CC mechanisms and as a measure of CC load on DDBMS resources. We also describe two new nonblocking distributed concurrency control mechanisms which use the concept of multiple data object versions. One is based on time stamp ordering of transaction execution and the other is based on nonserializable execution detection and recovery to serializable execution. We compare both with distributed two-phase locking.

## 1. Introduction

Over the last few years the importance of distributed DBMS's has been widely recognized. Consequently there has been considerable research on the most important aspect of distributed DBMS--concurrency control (CC). This paper argues that despite numerous papers on concurrency control [THO76, THO79, BER78, ASL76, STO78, BAD78, ELL77, LAM76, LIN79, KUN79, REE78, RIE79, MOL79, BAD79b, KAN79, LEL78, HER79] there are very few generic CC mechanisms or algorithms and as a result the majority of CC proposals are extensions, variations or modifications of these. This is not to say that such CC proposals are less original. What we are arguing here is that most CC mechanisms are dissimilar to a far less degree than they are similar and this fact then suggests that one should attempt to classify them and to compare the properties of each class.

1

In this paper we divide CC mechanisms into three classes. Our CC classification criteria are based on conventional operating system concepts of mutual exclusion and synchronization, the degree of concurrency and the transaction serializability enforcement policy concurrency control mechanisms use to guarantee that the interleaved and concurrent execution of transactions is the same as if the same transactions were executed in some serial order, i.e., one after another. Such policy can be to avoid, prevent, or detect and resolve nonserializable executions. By the degree of concurrency we mean the degree of concurrent execution of conflicting transactions. For example, if two executing transactions need to access at the same time a set of data objects, then they will conflict. In this scenario the degree of concurrency is the number of transaction concurrent actions allowed by the concurrency control mechanism on the data objects on which transactions conflict or interfere. More precisely, the degree of concurrency as defined in [BAD80] is an average number of data objects exclusively held by a transaction during its execution time. This definition reflects the fact that if transactions interfere over the set of data objects, then the number of interfering transactions which cannot execute concurrently is directly proportional to the number of data objects exclusively held by one transaction during its execution.

The second part of this paper describes two new distributed nonblocking CC mechanisms. We also derive CC overhead for each class of CC mechanisms. The CC overhead is defined in terms of synchronization messages and the resulting delay. We derive CC overhead for non-interfering transactions and two cases each of two conflicting transactions. We consider the analysis of two transaction conflicts an appropriate demonstration of the differences among CC mechanism classes. Moreover, some recent results [GRA80] indicate that the probability of three or more transactions conflicting at the same time is extremely low.

## 2. Classification of CC Mechanisms

There are a number of possible classifications of CC mechanisms and it is not easy to choose one. We consider here the classification introduced in [BAD79a] which is quite consistent with the traditional operating system concepts. We distinguish three basic classes of consistent CC mechanisms. (A consistent CC mechanism is serializable or results in database states identical to those due to some serial execution, called serialization order, of the same set of transactions.) The MES or mutual exclusion set class includes any CC mechanism that satisfies the following characteristics: transaction can execute only if it has an exclusive access, at some time t, to all data objects at which it writes and a shared access to all data objects it reads. In other words, concurrent execution of transactions is based on a mutual exclusion over the set of data objects accessed by one transaction. Two techniques employed to achieve mutual exclusion over the set of data objects are two-phase locking [ESW76, GRA78, STO78, ELL77], and sequence numbers (or time stamps) [THO79, THO76, ROS78]. Another

characteristic of MES class is that the serialization order is always determined at execution time and it cannot be a priori determined or guaranteed. MES class can be further divided by other classification criteria such as centralized or decentralized control and processing. Typical examples of MES class can be found in [STO78, GRA78, THO79, ROS78, ELL77, ALS76, MOL79, KUN79].

The second class of CC mechanisms is S or synchronization class. The usual technique to achieve synchronization involves the use of a unique sequence number (often called a time stamp) assigned to each transaction. The distinct property of S class CC mechanisms is that the transactions must execute in the order of their time stamps, and thus if necessary an a priori ordering of transaction execution can be guaranteed. Again, one could further classify S class according to the way sequence numbers are generated, whether the transaction can have its sequence number changed, etc. The typical representation of S class CC mechanisms include [LAM78, BAD78, LEL78, BER78, REE78, KAN79, HER79]. We note here that although the CC mechanism in [REE78] is based on time stamp order, in execution it is fundamentally different from other CC mechanisms. The S class of CC can be divided into two subclasses: strong and weak synchronization. The strong S (or SS) subclass [BAD78, KAN79] requires that transactions execute in the order of their original sequence numbers. This means that transactions should be rejected only because they violated integrity constraints. In another words, no transaction executing under SS class should be rejected due to synchronization. We believe a demand exists for a type of CC mechanism that can guarantee an a priori ordering of transaction execution. For example, most real time DBMS's, like air traffic control and command and control, would require strong synchronization. The weak synchronization [BER78, LAM78, LIN79] (or WS) subclass still requires the execution of transactions in the order of their sequence numbers but the sequence numbers can be reassigned. Thus transactions can be rejected because of synchronization or integrity constraints violation. Therefore, the order of transaction execution cannot be guaranteed. The SS subclass requires data object preclaiming, i.e., data objects are known and claimed before transaction execution; otherwise SS class will cause serial execution of all transactions. The WS subclass allows run time claiming of data objects.

The third class of CC mechanisms, called MEO is based on the mutual exclusion over one data object at a time and a set of sequencing rules. An example of MEO class CC mechanism can be found in [BAD79b] and in this paper.

The above classification scheme reflects the degree of concurrency and the degree of optimism about the probability of transaction conflicts, i.e., a way in which each CC class minimizes CC overhead associated with conflicting and nonconflicting transactions and in a way in which each CC class guarantees serializable (SR) execution. The MES class simply prevents non-SR executions by a pessimistic conflict resolution policy which

3

considers almost any interference as a source of non-SR execution. The S class also prevents non-SR executions by using a less rigid but still pessimistic conflict resolution policy (time stamp execution order). Finally, the MEO class allows non-SR execution to occur and then to recover to SR execution by using an optimistic transaction conflict resolution policy. Such policy is optimistic in a sense that it assumes that not only transactions conflict infrequently but also that many transaction interferences do not necessarily result in non-SR execution.

The MES class is the least optimistic and provides the lowest degree of concurrency, while the MEO class is the most optimistic and provides the highest degree of concurrency. In order to explain this clearly we use the following example. Let's consider two transactions T[i] and T[j] which arrived a short time apart and which access the same set of data objects 1, 2, 3 and 4. As shown in [BAD79b] the sufficient and necessary conditions for SR execution of transactions can be expressed in terms of sequencing the transaction actions on data objects they access. The execution of two interfering transactions is SR if T[i] and T[j] executed in the same order on all data objects on which they interfered in read-write or write-write manner. Now consider two cases of T[i] and T[j] execution. First suppose that T[i] must write in the order 1, 2, 3, 4 and T[j] in the opposite order. Then if T[i] and T[j] execute under the MES, MEO or S class of CC mechanisms, they can execute serially, i.e., one only after other terminated. However, if T[i] and T[j] access 1, 2, 3 and 4 in the same order, then MES class of CC will again force serial execution. The S and MEO class of CC would allow one transaction execution to follow another just one data object behind. However, this case could occur in the S class only if two conditions are satisfied. First, the sequence numbers i, j must differ by one increment, i.e., i<j or j<i and there is no sequence number k such that k<j and i<k or k<i and j<k. Second, the transaction executed later must have a sequence number larger that the preceding transaction. As this is not generally the case, the S class rule requiring transactions to execute in order of their time stamps forces any transaction accessing some data object to follow one of two rules: wait on accesses by all transactions with smaller sequence numbers, or access the object and then either reject any transaction with a smaller sequence number or, if the access by the smaller number is allowed, rollback.

Thus, although the S class of CC mechanisms in principle would allow the trailing execution of two transactions, it cannot do so fully because of the sequence number transaction execution rule and the uncertainty about adjacency of sequence numbers if they are generated at each site, i.e., in distributed manner. To explain this phenomenon in another way, in MES class CC the sequencing decision is essentially local to the interfering transaction, while in S class it can be either global to all transactions, as in [LAM76, BAD79, KAN79, LEL78, HER79], or partially localized, as in [BER78].

The MEO class of CC allows transactions to trail each other because their

4

interleaving is constantly checked for its serializability [BAD79b]. As in MES class, the sequencing decision is local to the interfering transactions only, and thus it is not affected by other transactions in the system.

## 3. Concurrency Control Overhead

In order to investigate CC overhead for each CC class we must do three things. First, define CC overhead. Second, choose or construct a representative CC mechanism for each class. Third, select some scenario. The scenario we will consider here is a partially relicated n-node DDBMS. We will assume that our hypothetical transaction running under CC class representative CC mechanism accesses e nodes for transaction execution and r nodes for the update of replicated data objects.

We define two types of CC overhead. One type, called CC no-conflict, is a constant overhead per transaction due to the CC mechanism. It has three inseparable aspects. One is CPU and I/O load at each node (due to CC information processing such as messages, locks or time stamps) and the network load (due to CC messages). The second aspect is a delay experienced by the transaction before CC mechanism allows it to execute. CC delay has two parts. One is the communication delay due to CC messages and their sequencing. The second part is due to sharing of DDBMS resources with other transactions or other processes. The second part of CC delay can be evaluated only by a simulation or possibly by a detailed analysis using standard queueing theory approach. This is so because the second part of CC delay is a function of several system and load parameters. However, the first part of CC delay is the function of CC mechanism only and can be easily established for most CC mechanisms. We will therefore consider the first part of CC delay only and from now on we refer to as the CC delay. The third aspect of CC no-conflict overhead is the number of CC messages (and their sequencing) needed to guarantee a robust and serializable execution of the transaction. The CC delay is intimately related to the number of CC messages and their sequencing. This paper considers only the CC messages and the associated delay as the measure of CC no-conflict overhead.

The second type of CC overhead, called CC conflict overhead is associated only with conflicting transactions and it consists of the same three aspects as the CC no-conflict overhead. Again as in the case of no-conflict CC overhead we consider CC conflict overhead only in terms of CC messages and corresponding delay. We consider CC conflict overhead for all three classes of CC mechanisms in two simple scenarios. Each scenario consists of two interfering transactions T[i] and T[j]. The transactions T[i] and T[j] access (or read and write) three data objects 1, 2 and 3 at nodes 1, 2 and 3. In scenario 1 they access 1, 2 and 3 in the reverse order and in the scenario 2 in the same order. In each scenario both transactions arrive a short time apart.

In order to analyze CC overhead for each class of CC mechanisms we must select representative for each class. For MES class we use distributed two-phase locking and for MEO class we use distributed CC mechanism described in [BAD79b]. For S class we analyze distributed nonblocking CC mechanism proposed in this paper.

We consider here CC mechanisms which produce serializable executions and have a minimum degree of robustness obtained by using two-phase commit (2PC) or its equivalent.

### 3.1  MES Class CC Overhead

A description of distributed two-phase locking (D-2PL) CC mechanisms has appeared in several papers [GRA78, STO78, RIE79, LID79, GRA80a] and we repeat the basic rules:

1.  each node has a concurrency controller managing data local to that site

2.  any transaction which reads data object can read only after it has placed read lock either on the unlocked data object or read-locked data object

3.  any transaction which needs to write on data object can do so only after it write-locked the unlocked data object

4.  any transaction can unlock any of its already read-locked or write-locked data objects only after it read-locked or write-locked all data objects needed for its execution

The transaction execution under D-2PL CC mechanism with centralized two-phase commit (2PC) has two steps:

1.  Transaction locks at e sites and executes at e sites

2.  Transaction coordinator sends r "lock and prepare to commit and update" messages and (e-1) "prepare to commit and to delete    lock" messages during the first phase of the 2PC and waits for acknowledgement.    During the second phase of 2PC the transaction coordinator sends (e+r-1) "commit (or abort) and    delete locks" messages. After all sites acknowledged previous messages the coordinator site commits (or aborts) and releases its locks.

Thus the CC no-conflict overhead for D-2PL is as follows:

CC delay = 4T

6

number of CC messages = 4(e+r-1)

where T is the average communication delay between one sender and several destinations.

As mentioned earlier, we consider CC conflict overhead for all three classes of CC mechanisms in two simple scenarios. We consider the conflict of two transactions only. The transactions T[i] and T[j] access three data objects at nodes 1, 2 and 3. We assume each access to be exclusive, i.e., at each node transactions read and write. In scenario 1 they access 1, 2 and 3 in the reverse order, and in scenario 2 in the same order. In each scenario both transactions arrive a short time apart. In scenario 2 the transaction which arrived later will have to wait and thus the D-2PL conflict overhead for scenario 2 consists of the delay 3DELTA T + 4T, where DELTA T is the average processing time at each node and T is the average delay between one sender and several destinations. The delay 4T is due to 2PC. In scenario 1 the CC conflict overhead, assuming a centralized deadlock detection and resolution, is as follows. First, the conflicting transactions must wait for some fixed period of time, say Tw, and then report to the deadlock detector (delay 2T) which resolves the deadlock by rolling back one transaction. Thus the CC conflict overhead in scenario 2 consists of delay Tw + 2T + TROL, where TROL is the transaction rollback time. Assuming centralized deadlock detection the number of CC messages is 4 (2 from each transaction to deadlock detector) + 2 (rollback of one transaction from one site when deadlock occurred at site 2). By averaging CC conflict overhead from both scenarios we obtain:

CC delay = 1/2(6T + 3 DELTA T +Tw + TROL)

number of CC messages = 3

## 3.2  S Class CC Overhead

The S class is not easy to analyze because two radically different time stamp based strategies can be used to keep database consistent. One generally accepted strategy is to execute updates as soon as possible so that the incoming transactions are not delayed. In another words, such strategy results in a continuous adjustments to keep database consistency. Most of S class CC mechanisms use this strategy. The second strategy proposed in [BAD78] is to insure database consistency whenever it is necessary, e.g., when read on data objects with a given time stamp is to be executed all updates on that data object with smaller time stamps are fetched and executed. Such strategy emphasizes the fact that it is not important that the database is consistent continuously all the time as long as it is guaranteed that each transaction executes on consistent data.

In this paper we will analyze only the first strategy. Even such analysis is difficult as there are some C class CC mechanisms which are up to some degree adaptive, i.e., their CC overhead can be decreased (or increased) for example, by distributing and clustering primary copies at different sites or by distributing data and concurrency control at different sites as in SDD-1 [BER78]. Since in the MES class of CC we have analyzed two-phase locking which does not require any a priori assumptions, as for example, a priori known set of transactions or guaranteed FIFO communication network protocol we limit S class analysis to CC mechanisms which also do not require any a priori restrictive assumptions. The S class CC mechanism we investigate is described for the first time in this paper. It is based on the concepts of data object logs as described in [BAD79b], multiple versions of data objects and the enforcement of time stamp ordering of transaction execution. The proposed algorithm allows transaction rejection (due to integrity constraint violation or due to synchronization) and its resubmission, and therefore it belongs to a WS subclass of CC mechanisms. The CC mechanism is made robust by using two-phase commit and it can be described as follows.

Each named data object (DO) in the database has associated with it a log, called DO log. DO log contains entries by each transaction which read or updated a given DO. DO log entry consists of transaction ID, its time stamp, the list of fields and records (or tuples and attribute fields) transaction read or updated, and the status of read or update (temporary, aborted, committed). Transaction generates DO log entry after it has executed access to a given DO and it deletes its DO log entries during the two-phase commit (2PC). CC mechanism described here is nonblocking as opposed to any CC in MES class which are blocking. That is to say in MES class CC mechanisms one transaction can block or prevent other transactions from accessing the data objects (DO) it needs. The general idea underlying proposed S class CC mechanism is that each transaction generates a new version of data objects it updated. Such versions are temporary until transaction is committed and then they become permanent. However, temporary versions are seen by any other transaction as new versions of data objects. Basic rule is that new temporary version of DO can become permanent only after the preceding temporary version becomes permanent. For example, if transaction T1 generated version DO[T1] of DO and T2 generated version DO[T2] of DO from DO[T1], then DO[T2] can become permanent only after DO[T1] becomes permanent. In other words, each transaction makes its output immediately available to any other transaction and therefore, it does not block other transactions. Since the execution of transactions must occur in time stamp order only serializable executions are generated.

After the transaction made an access to DO or its latest version, and generated DO log entry, the DO log algorithm pushes DO log entry onto DO log. DO log is stack-like structure with push operation only. Deletion of DO log entries can occur in any order. Any push operation triggers the following actions. New DO log entry is checked whether it conflicts with other DO entry below it in the DO log. If it does, the time stamps of DO

8

log entries are compared and out-of-time-stamp execution can be detected for update-update or update-read conflicts if the new DO entry has smaller time stamp. If out-of-time-stamp execution is detected, the transaction which generated the latest entry to DO log is rejected. This means that all its so far generated DO log entries are marked as aborted during 2PC. Consequently, any other transaction which used the output of aborted transaction will be aborted as well. However, if no out-of-time-stamp execution is detected, then DO log algorithm allows the transaction to proceed in its execution. After the transaction finished its execution, it will use transaction coordinator and 2PC to post the updates to replicated DO's as well as to check at DO logs of replicated DO's whether the updates are in the time stamp order. It will also check by the first phase of 2PC whether the preceding conflicting DO log entries are marked as commit or abort. The acknowledgement of the first 2PC message is generated only after preceding conflicting DO log entries are either comitted or aborted. After the acknowledgement transaction coordinator either aborts (if any preceding conflicting transaction aborted or if any site decides to abort this transaction) or it commits (if no out-of-time stamp execution is detected and all preceding conflicting transactions committed). If transaction commits, then its updates are made permanent. The same message from the coordinator (i.e., the 3rd message of 2PC) to all sites accessed by the transaction marks DO log entries as committed (or aborted). DO log algorithm responds to the third message of 2PC (commit) by checking whether there is any DO log entry (i.e., below or above in the stack) which conflicts with committed entry. If there is none, the committed entry is deleted (or marked as deleted if it is to be used for system recovery). If there is a conflicting entry, then the committed entry can be deleted only after the conflicting entries are marked as committed or aborted. Finally, all involved sites acknowledge the 3rd message of 2PC and the coordinator site deletes (or markes as deleted) it DO log entries. The DO log algorithm responds in the same fashion to "abort DO log entry" messages.

As can be seen from the description of this CC algorithm, the time stamps are being used for resolution of transaction conflicts and for serializability of conflicting transaction execution. The DO logs are dynamically changing and their size is proportional to the frequency of transaction conflicts. Described CC mechanism is optimistic one as it assumes that the conflicting transactions will generate an out-of-time-stamp execution with probability lower or at worst equal to the probability that they generate execution in the time stamp order. This implies that at worst case 50% of conflicting transactions will be aborted and executed serially (i.e., as if they executed under MES class of CC). However, at least 50% of conflicting transactions will execute in much shorter time because of nonblocking character of this CC mechanism. We want to point out that although the proposed CC mechanism is optimistic it is not completely optimistic because it uses time stamp ordering for transaction conflict resolution. This to say that not all out-of-time-stamp executions are necessarily nonserializable. For example, assume that transaction T1 updates DO's 2, 3 and 4, and T2 updates 1, 3 and 5. Suppose they

9

conflict in out-of-time-stamp order at 3. Because of time stamp order *execution rule*, T1 or T2 or both will be aborted even if their execution is serializable. Of course, if T1 and T2 executed at 3 in time stamp order, then they can execute concurrently.

Assuming the same T1 and T2 executing under MES class of CC then T1 or T2 will be blocked at 3 and will have to wait for at least $3T + DELTA\ T$. We note that the CC mechanism which is truly optimistic, i.e., one which is based on nonserializable detection *and recovery* has been proposed in [BAD79b] and is also described in this paper later on. If T1 and T2 should execute under such truly optimistic CC mechanism, they could execute concurrently regardless at what order they accessed data object 3.

Now we derive CC overhead for the proposed CC mechanism. No-conflict CC overhead is easily seen to be:

CC delay = 4T

*number of CC messages* = 4(e+r-1)

The conflict overhead for scenario 1 when T1 and T2 read and update and conflict at sites 1, 2 and 3 in the opposite order is as follows. Let's assume that T1 has smaller time stamp. Then T1 can detect out-of-time-stamp order execution at 1, 2 or 3, where detection at 1 or 3 are extreme cases. We consider therefore detection of out-of-time-stamp execution at 2 as an average CC overhead. When T1 reaches site 2 and detects out-of-time-stamp execution (i.e., T2 already made DO log entry at site 2) T1 is aborted by two-phase commit mechanism from site 2. T1 will have to be resubmitted with a new or the same time stamp.

If T2 generated new DO version at site 1 from DO version generated by T1, then T2 will abort when it attempts to commit. T2 then has to be resubmitted with a new or old stamp. Of course, T1 and T2 resubmission could lead to a cyclic restart and rejection. We assume here that some simple method can prevent such situation, e.g., the system can delay one transaction until the other one commits. However, if T1 is aborted at site 1 before T2 generates new version of DO from T1 output, then T2 can commit. Let's assume the former case and then CC delay is 3T (time T2 needs to detect that T1 aborted and to abort itself). The number of CC messages is 3 (due to T1 abort) + 6 (due to T2 abort).

The conflict CC overhead for scenario 2, when T1 and T2 read and write and conflict at sites 1, 2 and 3 in the *same order* is as follows. If T1, which has smaller time stamp, reaches site 1 before T2, then T2 can follow T1's execution one site behind. This means that T2 can commit immediately after T1 commits. The only CC overhead is the delay DELTA T experienced by T2. There are no CC overhead messages in scenario 2.

Averaging the CC conflict overhead from both scenarios we obtain:

CC delay = $1/2(3T + DELTA\ T)$

number of CC messages = $1/2(3 + 6)$

### 3.3 S Class Overhead Revisited

Comparing CC overhead of MES and S class we can see that they are quite similar. In particular for non-conflicting transactions, which are vast majority in most applications, the CC no-conflict overhead is identical. Of course, the main reason is the use of two-phase commit (2PC) for insuring the robustness. 2PC is a fault-tolerant communication protocol intended to tolerate some faults while still performing the intended function which is to ensure atomic property of one operation at different sites as, for example, update of multiple copies or release of locks or atomicity of transaction itself. Thus 2PC although not designed for or derived from the two-phase locking (2PL) is nevertheless very natural way of implementing robust 2PL. The point is that 2PL and other MES class CC mechanisms are blocking mechanisms when by locking some data object other transactions are blocked or prevented from accessing the same data object. Since 2PL is blocking it is important that once the transaction commits the locks are explicitly deleted as soon as possible and in a reliable fashion. 2PC serves very well that purpose. However, S and MEO classes of CC mechanisms are nonblocking and therefore, there is no pressing need to use 2PC in order to achieve the same degree of robustness. As a matter of fact the use of 2PC for nonblocking CC mechanisms is a major drawback for such mechanisms as 2PC negates their inherent advantages and makes them, at least in terms of CC overhead, equivalent to blocking CC mechanisms. Of course, the major advantage of nonblocking CC mechanisms is that they are nonblocking and therefore, there is no need to delete (or to mark as deleted) DO log entries (or other structures) used for serializability as soon as possible after transaction terminated. Notice that the proposed CC mechanism can use one structure, DO logs, for recovery and concurrency control as well. On the contrary, blocking CC mechanisms (NEO class) use two distinct structures--lock tables for concurrency control and logs for recovery.

Considering CC mechanism described in the previous section of this paper we will address the following problem. Can we modify this algorithm in such way that its robustness is preserved but its CC overhead is decreased by eliminating 2PC? The answer to this question is positive and we indicate here what modifications are needed. Consider the following modifications. Let's assume nonconflicting transaction Tn. Once Tn terminated execution, i.e., it did not execute out-of-time-stamp order at any DO it accessed, Tn instead of committing by 2PC its DO versions (as permanent DO versions) will just change its status at the site it entered and will exit the system (called initiating site) from executing to terminated. This can only happen if the initiating site knows that

11

conflicting preceding transactions already committed. This can be accomplished by CC overhead messages to such transactions initiating sites. Then Tn will use the ongoing network traffic to piggyback its "delete my DO log entries and commit my DO versions." For example, if later on some other transaction T1 (with larger time stamp) should interfere with Tn's not yet deleted DO log entry, T1 can interrogate Tn's initiating site (DO log entry now must contain the address of that site) about Tn's status. This can happen either when T1 "bumps" into Tn's DO log entry for the first time or after T1 terminated but before T1 can be released from the system. In another words, T1 has to know whether Tn terminated so that the DO versions it computed from Tn versions can be made permanent by piggybacking its "delete DO log entries and commit my DO versions." We note here that blocking CC mechanisms cannot use piggybacking of messages because locks must be deleted as soon as possible.

We now analyze 2PC protocol. A traditional concept of 2PC allows any site to abandon transaction which already executed at that site but it has not committed yet. The major reason for such abort by the site is blocking character of 2PL. In another words, as transaction already locked and executed at such site (and therefore, there is no reason to abort because of program execution failure at that site), then the only reason the site would want to abort is that the resources blocked by a given transaction have to be released. Therefore, in 2PC the first message to all sites involved in transaction execution is intended to verify that none of the sites unilaterally aborted transaction. Assuming that short duration site failures do not constitute the reason to abort the transaction, then the first phase of 2PC in 2PL is needed solely to verify that the transaction was not aborted at any site and that its resources at that site are still sequestered [LID79] or blocked. The second phase of 2PC is then intended to notify each site either to abort or to commit, i.e., to make transaction generated output available to user or end other transactions. (Good description of what types of transaction output should be released or deferred until commit can be found in [GRA80a].)

We shall now argue why the proposed CC mechanism does not require 2PC while still being robust. In the proposed CC mechanism transaction output becomes available immediately after the transaction executed at a given site. (In the proposed CC mechanism there is no equivalent of traditional 2PC commit point.) Moreover, since the proposed CC mechanism is nonblocking, i.e., it does not block site resources after transaction executed at that site, then there is no reason why the site should abort the transaction. Again we assume that site short duration failures do not constitute the reason to abort the transaction at that site. Thus the proposed CC mechanism assumes that once the transaction terminated successfully its execution, then in terms of 2PC all of its sites already agreed to commit. Therefore, the proposed CC mechanism must only guarantee that the second phase of 2PC is performed. This means that "delete (or mark as deleted) my DO log entries and commit (i.e., make permanent) my DO versions"

messages to each site involved in transaction execution are delivered reliably but <u>not</u> necessarily as distinct messages. Because of nonblocking character of the proposed CC mechanism such messages and their acknowledgements can be piggybacked on the ongoing network traffic. If the messages are piggybacked, then in the worst case the CC no-conflict delay is 2T and there are 2k messages, where k is a number of transactions which are terminated but whose DO entries were not deleted yet. In the best case there is no CC no-conflict delay and no CC messages. Assuming the best and the worst cases occur with the same probability then the average CC no-conflict overhead is:

CC delay = T

number of CC messages = k

If the messages are not piggybacked, then no-conflict CC overhead is:

CC delay = 2T + T = 3T

number of CC messages = 2(e+r-1) + k

CC conflict overhead for the modified version of time stamp based S class CC mechanism described in this section can be derived as follows. Consider scenario 1 when two transactions, say T1 and T2, read, update and conflict at sites 1, 2 and 3 in opposite order. Suppose that first out-of-time-stamp execution occurs at site 2. Then transaction which made detection will abort itself by changing its status at its initiating site to aborted. The second transaction when it terminates sends "what is your status" messages to the initiating site of transaction with which it as far as it knows conflicted in the time stamp order (i.e., one which precedes it in DO logs). Acknowledgement of such message in scenario 1 is "aborted" message and the transaction changes its status to aborted as well. Of course, the change of transaction status to aborted means that the aborted transaction will piggyback on ongoing network traffic "delete my DO log entries and my DO versions" messages to all sites where it executed. Assuming the above described sequence of events (i.e., T1 detects out-of-time-stamp execution and sends abort to its initiating site /delay T and 1 messages/; T2 executes at site 1 (or 3) resulting in delay DELTA T and then T2 exchanges 2 messages with initiating site of T1 /delay 2T/) the CC no-conflict overhead is:

CC delay = 3T + DELTA T

number of CC messages = 3

CC conflict overhead for scenario 2 is:

CC delay = DELTA T

number of CC messages = 0 (as both T1 and T2 terminate at the same site 3)

Averaging CC conflict overhead from both scenarios we obtain:

CC delay = 1/2(3T + 2 DELTA T)

number of CC messages = 1/2(3)

### 3.4  MEO Class CC Overhead

MEO class consists of nonblocking CC mechanisms. This means that their output is available to any other process during transaction execution. The MEO class CC mechanism we analyze here is described in [BAD79b] and it differs from the one described in section 3.3 of this paper in one major respect--it is not based on time stamp order of transactions execution. It is based on nonserializable execution detection and recovery to serializable execution. This gives the MEO class higher degree of concurrency because some out-of-time-stamp order executions which are serializable and which would be rejected by S class CC mechanisms can be realized under the MEO class of CC.

The algorithm can be described best by comparing it to the CC mechanism described in section 3.3 of this paper. Both CC mechanisms use DO logs. However, the MEO class algorithm detects nonserializable executions as follows. When transaction Tn made an access to DO it pushes DO log entry onto DO log. Such entry consists of Tn's unique I.D., Tn's initiating site (i.e., the site where Tn enters and exits the system), list of records and fields Tn read or updated, and their status (temporary, aborted, committed), and so far accumulated Tn's conflict history. DO log algorithm checks whether there is any DO log entry conflicting with new entry. If there is, then Tn creates its conflict history for a given DO. The conflict history is the list of conflicting transactions reads and updates in the same order as they are in DO log. At the next DO Tn deposits its so far accumulated conflict history and updates it from that DO. The idea is that as Tn hops from one DO (or site) to another it deposits at each DO its cumulative conflict history which says with what other transactions Tn conflicted, where and how (i.e., read-read, read-update, update-update).

Since every transaction generates its conflict history then if two transactions, say T1 and T2, conflict they can determine at once, or when both terminated, whether they generated nonserializable execution. To explain this consider T1 and T2 updating DO's 1, 2 and 3 in the same and opposite orders. As long as, say, T1 precedes T2 in any update-update, or read-update conflict at all DO's, then the execution is serializable. If T1 and T2 execute at 1, 2 and 3 in the same order, then both can immediately detect nonserializable execution. Consider the following scenario. T1 updated 1 before T2.

14

However, at 2 T2 got ahead of T1 and updated 2 before T1. At this point T1 can decide from T2's conflict history (which is a part of T2 DO log entry at 2) that T1 and T2 generate nonserializable execution. T1 can also decide from its and T2's conflict history what is the best way to restore serializable execution. In our scenario T1 sends T2 "roll back up to 2" message. When T2 reaches 2 T1 and T2 can resume execution at 2 in correct order.

However, if T1 and T2 execute at 1, 2 and 3 in the opposite order, then T1 and T2 can detect their nonserializable execution only after they terminated as follows. After T1 and T2 terminated they both know that they conflicted in a serializable way in 2 DO's (either 1, 2 or 2, 3). However, T1 neither T2 know whether they conflicted at the 3rd DO. One way they can find out is by exchanging their conflict histories at their initiating sites (where they return after the computation). This exchange will enable their partial roll-back and recovery to serializable execution. (More detailed description of CC mechanism behaviour when more than two transactions conflict can be found Appendix or in [BAD79b].)

Of course, another way to find out whether T1 and T2 generated nonserializable execution is by using two-phase commit when upon termination each transaction would check at each site for nonserializable execution and for termination (or commit) of preceding interfering transactions. For example, consider T1 and T2 executing in the opposite order. Let's assume that T1 and T2 do not generate their conflict histories. When T1 or T2 terminate they can make their temporary version permanent by using 3 messages of two-phase comit protocol. That is to say the inititating site of T1 or T2 sends one message to each site it accessed. Such message is acknowledged and a relevant subset of DO log is returned also. Then the initiating site can decide whether its transaction a) generated nonserializable execution, b) has to wait for termination of preceding transaction in order to determine serializability of its execution.

Assume that T1 terminates first and tries to commit. From the first message of 2PC and its acknowledgement T1 can determine that it conflicted with T2, i.e., T2 preceded T1 at some DO. Therefore, before T1 can make its output permanent, it must wait for T2 to make its output permanent. However, T2 when it attempts to commit will detect nonserializable execution and will initiate recovery to serializable execution. CC overhead for 2PC variation of MEO class CC mechanism is essentially identical to the CC overhead of time stamp based S class CC mechanism described in section 3.2.

We want to emphasize that T1 and T2 can detect nonserializable execution in two equivalent ways. One is during 2PC and the other is by a) transaction conflict history mechanism, and b) by communication between transaction initiating sites. Obviously in terms of CC overhead the second way is much more effective.

2PC is used to make the update of multiple copies to appear as an atomic action, i.e., either all updates are installed or none. In the CC mechanism proposed in [BAD79b] and also described here, the transaction FORK operation, as multiple copy update, is seen as a FORK of transaction process which then must JOIN at transaction initiating site. There the transaction can decide whether all updates have been posted (as temporary ones) and whether it generated serializable execution (either immediately or after waiting for preceding conflicting transaction(s)).

We now derive CC overhead for the S class CC mechanism proposed in [BAD79b] and also described in this section. The mechanism uses DO logs, transaction conflict histories, initiating sites communication, deletion of DO log entries and commit of temporary DO versions by piggybacking on ongoing network traffic. The CC no-conflict overhead is in the worst case 2T and in the best case none. 2T is due to "virtual" conflicts when executing transaction "bumps" into undeleted DO log entries of k terminated transactions (2k messages). Assuming the best and the worst cases to occur with the same probability, then the average CC no-conflict overhead for MEO class is:

CC delay = T

number of CC messages = k

MEO class CC conflict overhead for scenario 1 is as follows. T1 and T2 reading, updating and conflicting at sites 1, 2 and 3 in opposite order will have to terminate first before detecting nonserializable execution. Assume that T1 terminated first and T2 during T. Then the detection of nonserializable execution occurs by T1 talking to T2 initiating site after T2 terminated and by inspecting each other's conflict histories. Resulting CC delay is 2T and 2 messages are involved. Scenario 2 generates CC delay DELTA T and no messages. By averaging the CC conflict overhead from both scenarios we obtain:

CC delay = T + 1/2 DELTA T

number of CC messages = 1

4. Conclusion

In this paper we have analyzed three distributed CC mechanisms belonging to three different CC classes in terms of CC overhead, i.e., the number of CC messages and corresponding delay. We have also shown that they differ in the degree of concurrency they provide. We can conclude that in terms of CC overhead and degree of concurrency the nonblocking CC mechanisms outperform blocking CC mechanisms, or in another words, MEO class outperforms S class which outperforms MES class. However, the results

16

derived in this paper, although useful for CC mechanisms comparison, must be interpreted within the distributed database system and application parameter space as done in [BAD80a, MOL79, RIE79]. This is to say that although CC is the most important mechanism of distributed DBMS the derived results should not be interpreted as an absolute indication of distributed DBMS performance. For example, even if MEO class provides the lowest CC overhead the distributed DBMS might perform better under another CC mechanism for some applications or networks. In other words, as indicated in [BAD80a] each class of CC mechanisms might be most suitable for certain classes of applications and DBMS system parameters.

## 5. Acknowledgement

## References

[ALS76] Alsberg, P. et al. "Multi-copy resiliency techniques," Center for Advanced Computation, Report CA 6202, University of Illinois, Urbana-Champaign, May 1976.

[BAD78] Badal, D. Z. and Popek, G. J. "A proposal for disttributed concurrency control for partially replicated distributed databases," Proc. of the 3rd Berkeley Conference on Distributed Data Management and Computer Networks, August 1978.

[BAD79a] Badal, D. Z. "Concurrency control and semantic integrity enforcement in distributed databases," Infotech State of the Art Report on Distributed Databases, Infotech 1979.

[BAD79b] Badal, D. Z. "Correctness of concurrency control and implications in distributed databases," Proc. of COMPSAC 79, Chicago, November 1979.

[BAD80] Badal, D. Z. "On the degree of concurrency provided by concurrency control mechanisms for distributed databases," Proc. of the Inter. Symposium on Distributed Databases, Paris, France, March 1980.

[BAD80a] Badal, D. Z. "The analysis of the effects of concurrency control on distributed database system performance," Proc. of the 6th Intern. Conference on Very Large Data Bases, Montreal, October 1980.

[BER78] Bernstein, P. A. et al. "The concurrency control mechanism of SDD-1: A System for Distributed Databases," IEEE Transactions on Software Engineering 4, 3(May 1978).

[ELL77] Ellis, C. "A robust algorithm for updating duplicate databases," Proc. of the 2nd Berkeley Workshop on Distributed Data Management and Networks, May 1977.

[ESW76] Eswaran, K. P. et al. "The notions of consistency and predicate locks in a database system," CACM 19, 11 (November 1976).

[GEL78] Gelenbe, E. and Sevcik, K. "Analysis of update synchronization for multiple copy data bases," ibid BAD78.

[GEL79] Gelenbe, E. and Sevcik, K. "Analysis of update synchronization for multiple copy data bases," IEEE Transactions on Computers 28, 10 (October 1979).

[GRA78] Gray, J. "Notes on database operating systems," IBM Research Report RJ 2188, February 1978.

[GRA80] Gray, J. Personal communication.

[GRA80a] Gray, J. "A transaction model," in Automata, Languages and Programming, Lecture Notes in Computer Science 85, Springer-Verlag, 1980.

[HER79] Herman, D. et al. "An algorithm for maintaining the consistency of multiple copies," ibid KAN79.

[KEL73] Keller, R. M. "Parallel program schemata and maximal parallelism," JACM 3 (July 1973) and JACM 20 (October 1979).

[KUN79] Kung, H. T. and Robinson, J. T. "On optimistic methods for concurrency control," Proc. of VLDB Conference, Rio de Janeiro, Brazil, October 1979.

[LAM78] Lamport, L. "Time, clocks, and the ordering of events in a distributed system," CACM 21, 7 (July 1978). March 1976.

[LEL78] LeLann, G. "Algorithms for distributed data-sharing systems which use tickets," ibid BAD78.

[LID79] Lindsay, G. B. et al. "Notes on distributed databases," IBM Research Report RJ 2571, July 1979.

[LIN79] Lin, W. K. "Concurrency control in a multiple copy distributed database system," ibid BAD78.

[MIN78] Minoura, T. "Maximally concurrent transaction processing," ibid BAD78.

[MOL79] Garcia-Molina, H. "Performance of update algorithms for replicated data in a distributed database," Ph.D. dissertation, Dept. of Computer Science, Stanford University, June 1979.

[PAP79] Papadimitriou, C. M. "Serializability of concurrent database updates," JACM 26, 4 (October 1979).

[REE78] Reed, D. P. "Naming and synchronization in decentralized computer systems," MIT/LCS/TR-205, MIT, Laboratory for Computer Science, September 1978.

[RIE79] Ries, D. R. "The effects of concurrency control on database management system performance," Ph.D. dissertation, Computer Science Dept., University of California, Berkeley, April 1979.

[RIE79a] Ries, D. R. Personal communication.

[ROS78] Rosenkrantz, D. J. et al. "System level concurrency control for distributed database systems," ACM TODS 3, 2 (June 1978).

[STO78] Stonebraker, M. "Concurrency control of multiple copies of data in distributed INGRES," ibid BAD78.

[THO76] Thomas, R. "A solution to the update problem for multiple copy data bases which use distributed control," BBN Report 3340, July 1976.

[THO79] Thomas, R. "A solution to the concurrency control problem for multiple copy databases," ACM TODS 4, 2 (June 1979).

## Appendix

In order to demonstrate the behaviour of CC mechanism when more than two transactions conflict, let's consider an example of three conflicting transactions. Assume that T1 reads and updates at sites 1 and 2, T2 reads and updates at 2 and 3, and T3 reads and updates at 3 and 1. The transactions arrive short time apart and they conflict at each site they accessed as follows. At site 1 T3 precedes T1, at site 2 T1 precedes T2, and at site 3 T2 precedes T3. This means that T1 knows from its conflict history that upon its termination it should send its conflict history to the initiating site of T3. Similarly, T2 and T3 send their conflict histories to the initiating sites of T1 and T2. Each initiating site now constructs a precedence relation and checks it with other initiating sites. At that time the nonserializable execution is detected because the precedence relations will be inconsistent. In our example, T1 initiating site after receiving T2's conflict history knows that T3 precedes T1 precedes T2. The initiating site of T2 knows that T1 precedes T2 precedes T3 and the initiating site of T3 knows that T2 precedes T3 precedes T1. In the next step of initiating site communication T1, T2 and T3 independently detect nonserializable execution which is due to a cycle of conflicts. Now the cycle must be broken in order to recover to serializable execution. In our example the initiating site of T1, T2 and T3 have the same, and complete, information about the conflict cycle to make independently the same decision – to rollback. In order to avoid cyclic restart and rollback, they can restart at different time, i.e., with different delay. Such decision can be again made by each transaction independently by using, perhaps, their ID's. This example shows how the described CC mechanism would cope with a highly unlikely situation of three (or more) transaction conflicts.

INITIAL DISTRIBUTION LIST

| | |
|---|---|
| Defense Technical Information Center<br>Cameron Station<br>Alexandria, VA 22314 | 2 |
| Dudley Knox Library<br>Code 0142<br>Naval Postgraduate School<br>Monterey, CA 93940 | 2 |
| Office of Research Administration<br>Code 012A<br>Naval Postgraduate School<br>Monterey, CA 93940 | 1 |
| Chairman, Code 52Bz<br>Department of Computer Science<br>Naval Postgraduate School<br>Monterey, CA 93940 | 30 |
| D. Z. Badal, Code 52Zd<br>Department of Computer Science<br>Naval Postgraduate School<br>Monterey, CA 93940 | 10 |
| Robert B. Grafton<br>Office of Naval Research<br>Code 437<br>800 N. Quincy Street<br>Arlington, VA 22217 | 2 |